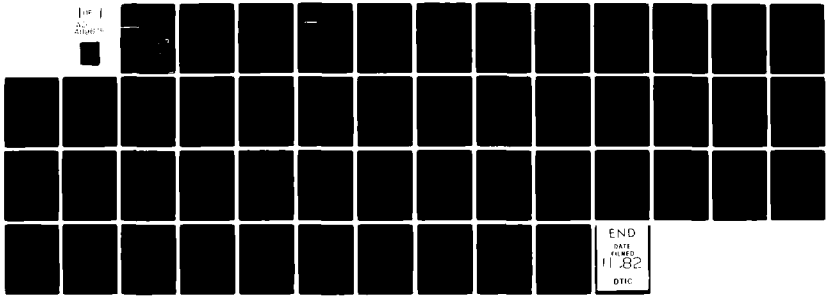
AD-A119 675 UNIVERSITY OF SOUTHERN CALIFORNIA MARINA DEL REY INFO--ETC F/G 9/2
PROGRAM DEVELOPMENTS: FORMAL EXPLANATIONS OF IMPLEMENTATIONS.(U)
AUG 82 D S WILE NSF-MCS79-18792
UNCLASSIFIED ISI/RR-81-99 NL



END
DATE FILMED
11-82
DTIC

ISI/RR-82-99
August 1982

David S. Wile

Program Developments: Formal Explanations of Implementations

INFORMATION SCIENCES INSTITUTE
4676 Admiralty Way/Marina del Rey/California 90291
(213) 822-1511
UNIVERSITY OF SOUTHERN CALIFORNIA

David S. Wile
Program Developments: Formal Explanations of Implementations
ISI/RR-82-99

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: ISI/RR-81-99	2. GOVT ACCESSION NO: A119675
	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)*: Program Developments: Formal Explanations of Implementations	5. TYPE OF REPORT & PERIOD COVERED: Research Report
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): David S. Wile	8. CONTRACT OR GRANT NUMBER(s): MCS-7918792
9. PERFORMING ORGANIZATION NAME AND ADDRESS: USC/Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, CA 90291	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: National Science Foundation, 1800 G St. N.W., Washington, D.C. 20550	12. REPORT DATE: August 1982
	13. NUMBER OF PAGES: 51
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*: ----------	15. SECURITY CLASS. *(of this report)*: Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

This document is approved for public release and sale; distribution is unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

---

18. SUPPLEMENTARY NOTES

---

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

program design, program development, program optimization, program transformation, programming environments, replay, structure editors

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

(OVER)

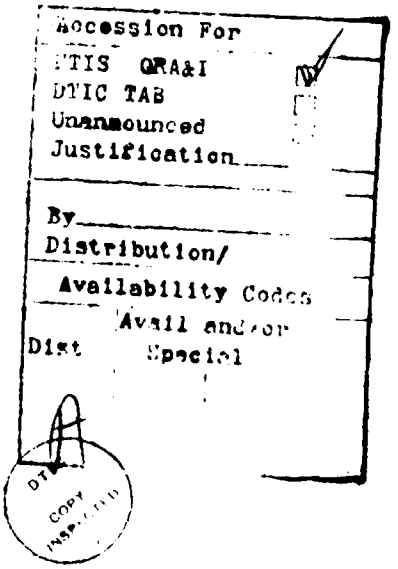
DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

20. ABSTRACT

Automated program transformation systems are emerging as the basis for a new programming methodology in which high-level, understandable specifications are transformed into efficient programs. Subsequent modification of the original specification will be dealt with by reimplementation of the specification. For such a system to be practical, these reimplementations must occur relatively quickly and reliably, in comparison with the original implementation. We believe the reimplementation requirement necessitates that a formal document--the program development--be constructed during the development process to explain the resulting implementation to future maintainters of the specification. The overall goal of our work has been to develop a language for capturing and explaining these developments and the resulting implementations. This language must be capable of expressing: the implementor's goal structure, all program manipulations necessary for implementation and optimization, and plans of such optimizations. In this report, we discuss the documentation requirements of the development process and then describe a prototype system for constructing and maintaining this documentation information. Finally, we indicate the many remaining open issues and the directions to be taken in the pursuit of solutions.



ISI/RR-82-99
August 1982

David S. Wile

Program Developments: Formal Explanations of Implementations

INFORMATION SCIENCES INSTITUTE
UNIVERSITY OF SOUTHERN CALIFORNIA
4676 Admiralty Way/ Marina del Rey/ California 90291
(213) 822-1511

This research was supported by the National Science Foundation under Contract No. MCS-7918792. Views and conclusions contained in this report are the author's and should not be interpreted as representing the official opinion or policy of NSF, the U.S. Government, or any person or agency connected with them.

CONTENTS

ACKNOWLEDGMENTS

Many thanks to Bob Balzer, Steve Fickas, Susan Gerhart, Neil Goldman, and Bill Swartout for their helpful comments on early drafts of this report. I would also like to thank Martin Feather and Phil London for helping to debug the Paddle system (as guinea pigs, of course). Most of the ideas in this report arose from discussions with these individuals and Don Cohen and Lee Erman. Thanks to Nancy Bryan for her critical corrections to the final draft of this report. And finally, particular thanks to a *CACM* referee who suggested major structural improvements to this report.

1. INTRODUCTION: TRANSFORMATIONAL IMPLEMENTATION

The programming paradigm considered here involves implementing a very high-level specification through the use of **correctness-preserving transformations**. The implementor--a person--chooses different transformations on the basis of his knowledge of the domain in which the program will ultimately run and his knowledge of their appropriateness. The computer actually applies the transformations and displays the results so he can then consider further transformations.

These transformations accomplish two separate tasks [Neighbors 80]: implementation--selecting realizations of abstract constructs in terms of more concrete ones, and optimization--rearranging a set of operations so as to minimize their execution cost. To get around the confusion between implementation of the specification and optimization of the implementation in the programming language, it has become common to talk simply of "optimization of programs" in a "wide spectrum language" [Bauer 81]. Such a language encompasses both specifications and programs. To do so, every construct must be operational, i.e., even the highest level constructs are executable (though very inefficiently). Hence, all transformations are potential optimizations. Throughout this report, we will tend to call the person performing the optimization and implementation the "implementor"; his task is "implementation of specifications" or, equivalently, "optimization of programs."

Although proponents of this paradigm have been active for several years [Balzer 76, Bauer 76, Burstall and Darlington 77, Loveman 77, Standish 76], no production-level system for transformational optimization has been designed [Partsch 81]. Several problem areas for the paradigm have become evident:

- Constructing a library of transformations that adequately captures most useful optimizations (for any specification/programming language). Standish [Standish 76], Barstow [Barstow 79], and Rich [Rich 81] have done pioneering work in this area.
- Indexing such a library so that one can browse through it to find transformations suitable to the purpose at hand. This is an essential component recently considered by [Neighbors 80] as a classification issue. A different approach to the problem is to develop **generic** transformations, encapsulating some large chunk of knowledge about several different but related transformations.
- Verifying and validating transformations to be correctness preserving. Work by Gerhart [Gerhart 75] and Broy and Pepper [Broy 81] has provided a technology for transformation verification, though its adequacy has yet to be tested on any significant set of transformations.
- Designing a mechanism for dynamically verifying that conditions in the program pertain, enabling the application of transformations. In its worst guise, this is the automatic theorem-proving problem; it may suffice to use flow-analysis techniques developed for traditional compilers (see [Geschke 72] and [Babich 78]) along with specialized **predicate pushing** mechanisms developed in program verification efforts (see [Deutsch 73] and [Dijkstra 76]) and transformation system designs (see [Cheatham 79]).
- Automating large parts of the transformation process. Enormous chains of primitive transformation applications are necessary to optimize even the most trivial specifications. **Simplification** [Kibler 78] and **conditioning** [Fickas 80] (getting the program into shape for a desired transformation) are two approaches to this problem. These are tied together by the work of Feather [Feather 79], in which the implementor describes how he would like the resulting program to look, along with some key (insightful) transformations the

system should use in obtaining it. Naturally, all work on optimizing compilers is relevant here [Schwartz 75, Wulf 75, Allen 75].

- Describing what the implementor did in optimizing the program--i.e., describing the design decisions as well as the particular steps he went through in producing the final program. Such information must be available for modifiers of an optimization design to be able to maintain the original specification. Feather [Feather 79], Feather and Darlington [Darlington 79], and Sintzoff [Sintzoff 80] have laid the groundwork for this largely unexplored problem.
- Scaling up--problems of **size**. For realistic applications, enormous numbers of transformations, transformation applications, intermediate program states, intermediate predicate states, etc., must be dealt with quickly. This makes size the most crucial problem to be solved.

However, not all of the problems need to be solved to obtain a useful, albeit incomplete, system. People currently maintain predicates correctly (or approximately correctly) with considerable success. Thus, it does not appear that proofs of programs or transformations are crucial; we can (temporarily) continue to rely on people to perform these tasks informally. Also, it is quite reasonable to expect that the automation problems--simplification and conditioning--will become more tractable and that an acceptable level of automation can be achieved through techniques such as preprocessing sets of transformations, using, for example, the ideas of Kibler [Kibler 78] and Knuth and Bendix [Knuth 70]; automatic data structure optimization, as begun by Low [Low 74]; and automation of conditioning transformations, as begun by Feather [Feather 79] and developed by Fickas [Fickas 80]. While these capabilities are being developed, a useful transformation system must rely on more intervention by the user. Hence, arriving at a useful, large catalog of transformations, supporting its perusal, and documenting the development process itself seem to be the unsolved problems most critical to realizing a practical transformation system.

This report focuses on the last of these problems--documenting the development of the optimization for the purposes of maintaining the specification and subsequently reimplementing it. What we call a **program development** is a formal document explaining the implementation of a specification for subsequent use by maintainers. The efforts of Feather and Darlington [Feather 79, Darlington 79, Darlington 82] expose the fundamental principle: if the application of transformations is expressed in a way that captures the **structure** or **optimization strategy** being pursued, it may be read later to understand how subsequent specification changes might impinge on the original optimization, and whether or not the original implementation strategy is still valid. N.B.: Feather and Darlington made the crucial observation that a **formal** structure representing the optimization has the potential to be **replayed** automatically--reapplied to a changed specification. Sintzoff [Sintzoff 80] defines precisely the notion of design decision and develops several commonly used structuring facilities. Cheatham [Cheatham 81] provides a mechanism for replaying the **historical** development of the program on subsequent versions. Swartout [Swartout 81] has designed a system to generate explanations automatically, given appropriate formal documentation of the primitives from which a program is constructed and the goals which they accomplish. The relationship of these bodies of work to ours will be detailed in the relevant sections which follow.

Our work concerns the nature of the formal object we call the **development structure**, which is **applied** to specifications to produce implementations. This characterization of development structures as objects applied to programs to produce other programs allows our development structures to encompass the related notions of developments, strategies, transformations, and editors. Transformations obviously satisfy this characterization. Editors are simply programs (usually

interactive) for applying sequences of transformations (not necessarily equivalence- or correctness-preserving). Strategies represent the intent, or plan, behind such sequences, and developments are the combination of all of these capabilities into a coherent structure.

Below we list the properties required of a development description language and relate this language to the development process itself and its use in replay. We then describe **POPART** [Wile 82], a prototype system we have built for experimenting with developments, transformations, and other program manipulations.

2. DEVELOPMENT LANGUAGE PROPERTIES

Recall that the principal reason for the development language is to enable future (re)implementors to understand how the original implementation was made. This does not actually necessitate a *formal development language in and of itself.* (As we mentioned above, the desire arises from the observation that developments expressed in a formal language could be reapplied to changed specifications (automatically) and, in some cases, would produce an appropriate reimplementation.) Hence, it is not the formal properties of the language that determine the desiderata for the development, but rather those properties of the language that will allow suitable explanation for reimplementation purposes.

In particular, the primary property of the development language is that it should allow the optimizer (human) to explain well (to the human reimplementer) the motivations and design decisions made in the original development of a program. At the very least, this implies a **structuring** of goals and explanations into goals with subordinate goals or ways of achieving them. Hence, some mechanism for **subordination** will be required: traditional mechanisms achieving these include **named subfunctions** and **explicit refinements** ("do *X* by doing *Y* and *Z*").

In addition, goals at the same conceptual level must be related to one another; hence, the need for mechanisms conveying **goal dependencies** as perceived by the implementor. For example, it will certainly be essential to understand that two subgoals are independent. The maintainer should be able to ignore independent subgoals that deal with sections of the specification unaffected by a change.

The particular goal structures we have foreseen include the following:

- Sequential dependency (composition): Goal *A* must be achieved before goal *B*.
- Goal independence: Goal *A* may be achieved in parallel with goal *B*.
- Choice: Goal *A* was chosen from a set of possible goals $\{A, B, ...\}$ all of which supported the same overall goal.
- Conditional goals: Goal *A* need only be achieved if *B* could not be achieved.
- Repetitive goals: achieve a set of goals $\{A_1, A_2, A_3...\}$.

Other primitive goal structures may become important as we gain more experience with developments and development languages.

More complex goal structures should also be expressible and, most importantly, **definable**, for these correspond to **plans** or **strategies** in design activities. Certainly they need to be parameterizable as well. We see a spectrum of plan-like objects, spread along the axis of "completeness" or "degree of parameterization." In particular, a low-level, single-purpose transformation is a complete plan that is quite certain to succeed with little intervention from the implementor. Transformations with "free parameters" are a little less transformation-like and a little more strategic--for example, a transformation that introduces an arbitrary predicate to break out a special case. At the "less complete" end of the spectrum, a plan for "divide and conquer" that reads "split off parts, apply function to parts and then combine results" is a highly parameterized, incomplete plan. It is clear that the implementor must be able to define and invoke the whole spectrum of plan types.

Interestingly enough, Sintzoff has independently arrived at essentially the same goal structuring facilities for recording design decisions. He includes an *inductive* decision type; we substitute several forms of conditionality (including loops and recursive plan invocation) to achieve the same ends. It would be surprising if any great differences were exhibited in such a minimal-semantics, decision-structuring language! The main source of difference lies in the primitives filling the structure and the interpretation of the structure.

Until now the discussion has not required any properties of the development language dependent on the choice of specification/programming language. Appropriately so, for the whole notion of implementation strategy and documentation is primarily language independent, relying only on "programming knowledge" (currently) locked inside experts' heads. The only real constraint on the development language that relates to the programming language is that all commands necessary to describe program manipulation be expressible in the language. This requires that the development language be grounded in some language for manipulating programs and program properties. If an extremely powerful underlying mechanism were present, this could be as simple as the single command "*achieve* goal." For our experiments we have chosen a quite basic editing language, but other quite different (primitive) languages could have been chosen and used successfully within our development language.

To summarize, the overall goal is to explain the implementation, using a formal development language which is capable of expressing: a rich goal structure, **all** program manipulations necessary to optimization, and plans for optimization as well as detailed optimizations.

3. OUR DEVELOPMENT LANGUAGE

The development language we have designed and implemented (called **Paddle**[1]) primarily emphasizes **structure**. The structural aspects of the language seem almost independent of the programming/specification language whose objects are being transformed. That independence is emphasized below, where the structural facilities are introduced first, followed by the actual primitives that manipulate the specifications.

---

[1] From **P**op**A**rt's **D**evelopment **L**anguage: a homonym.

3.1. Definition and Refinement

The need for definition facilities for transformations, strategies, plans, etc., was mentioned above. Although there may be strict distinctions between these various definable entities, we are not yet sure where to draw the boundaries. Hence, our development language currently supports only a single definition facility: the **command**, consisting of a name, a set of parameters and a body. Let's define the well-known problem-solving paradigm "divide and conquer." We would like to capture the essence of this paradigm in an abstract plan. We begin by defining the following Paddle command:

---

command DivideAndConquer(function,set) =
  begin
    split set into subsets, s_1, s_2, ...;
    compute a related function, f_1, on the subsets;
    combine values of f_1 on subsets via a new function, f_2;
    note You must insure that function applied to set =
        f_2 applied to $\{f_1(s_1), f_1(s_2), \ldots\}$
  end

The **begin/end** pair indicates the **sequential composition** of goals t tisfied by the implementor, i.e., the goals must be satisfied in the order stated. "Split," "com and "combine" are not understood by the development system as predefined commands. Rather, the user must **refine** these "stubs" to deal with the situation at hand when the command is actually invoked. In particular, we could refine the "split" stub into a binary decomposition of the set using the following syntax for **refinement** (a refinement is simply an in-place definition):

---

split set into subsets s_1, s_2, ...
  by
    binary partitioning into $s_1=\{e_1 \ldots e_{k/2}\}$ and
      $s_2=\{e_{k/2+1} \ldots e_k\}$;

The use of the reserved word **by** indicates that what follows the description is what was actually meant by the commentary before it. This will be indented and, thus, appropriately subordinate to the concept it implements. Thus, as with Caine and Gordon's Program Design Language [Caine 75], our development language provides a skeletal structure for English description leading ultimately to primitive Paddle commands.

3.2. Goal Structures

The examples above already illustrate two different goal structures: **sequential composition** and **refinement subordination**. Another goal type that arises frequently is an **and** goal: the optimizer wishes to convey independence of subgoals. Paddle allows this using the **each** construct. There are at least two varieties of **and** goal: each must be **achieved** independently or each must be **achievable** independently (but the order chosen may be relevant). The latter interpretation has been adopted in Paddle; the former may have to be introduced later.

Another Paddle goal type is a **choice** goal. For example, the transformation/plan designer may wish to convey more information about the alternative possible methods for doing the split above. This is accomplished using the **choose from** construct:

```
split set={e1,e2,...} into subsets s1, s2, ...;
  by
    choose from
       partitioning into s1={e1} and s2={e2,e3,...};
       binary partitioning into s1={e1...ek/2} and
                                s2={ek/2+1...ek};
       basis partition s0, s1, s2, s4, ... s2i,
          where each en is a linear combination of the s2j
    end
```

Presently, such choices are not made automatically; the implementor decides in each situation what the appropriate selection should be.

Conditional structures, are used to make automatic choices. Currently the only conditional structure, **first of**, is like LISP's COND, in which the first goal to succeed is the one chosen.[2] For example, the following indicates successively worse implementations (slower or requiring more space) for sequences:

```
first of
  ArrayImplementation;
  LinkedListImplementation;
  DoublyLinkedListImplementation;
  HashedImplementation
end
```

If each of the "Implementations" is a transformation, then the first one to be usable in the current situation will be the goal achieved. The failure of each goal is the conditionality for the attempt of the next alternative.

Finally there is a loop goal structure that enables the body to be executed (achieved) repeatedly **while** (or **until**) another goal is satisfied. An example of such a loop structure is one that implements all sets by repetitively applying the above conditional set implementation transformation tc each *unimplemented* set.

These goal structures provide Paddle with a general programming capability so that arbitrary developments can be constructed. Our goal is to make such developments both convenient and understandable.

---

[2] A syntactic variant, in the form of an if-then-else statement, is also planned.

3.3. Relationship with the Program Manipulation System

Paddle is a language for structuring goals. How these goals are achieved is an orthogonal issue dependent entirely on how the terminal nodes of the goal structures are defined. As we mentioned earlier, a single primitive command, **achieve**, could be used as the terminal node for all goal statements, which would leave to the transformation system the choices of how to achieve the primitive goals. We could be slightly less ambitious and allow "hints" to the transformation system by introducing Feather's **using** statement:[3] the goal is to be accomplished automatically, but it must **use** a set of named transformations in its achievement [Feather 79].

Alternatively, a large set of primitive commands could be used to describe very particular ways of achieving goals; some may appear to be actions rather than goals. Although all of these options are acceptable, we have chosen the last, in the form of an editing language, as the primitive Paddle node language expressing how to accomplish the goals stated in the development. This choice was thought to be both universal and easily implementable; further, as higher levels of automation are achieved (as is planned), more abstract "primitives" can be added. In the meantime, we will continue to have a functioning, usable system.

3.4. Operational Interpretation

In fact, the set of primitive commands is actually a parameter to Paddle; however, the fact that the goal structure is given an **operational interpretation** is fixed and crucial to the actual kinds of problem solving/design activity that can be expressed. In particular, the overall model of program manipulation used by Paddle is as follows: there is at all times a specification/program affected by Paddle expressions. This specification/program, together with the active goal structure(s), forms the data and control portion of the **state** of the "abstract Paddle machine." The development structure is applied to an initial state to produce a new state. That application is a relatively straightforward interpretation of the development language as though it itself were a programming language. In particular, it is a depth first, left-to-right tree traversal of the goal structure represented by the development.

The state into which the initial state is transformed depends on whether the development process contains any errors or is incomplete. In such situations, the new state represents "progress so far"; facilities are provided for fixing the Paddle "program" and continuing. When there are no errors, the development is entirely automatic, and the Paddle program indeed represents the entire implementation history of the specification: the final state is the implementation. N.B.: It is the **automatic** application of a development structure to a specification to yield the final implementation which guarantees the fidelity of the implementation explanation.

We emphasize that Paddle is executed as a programming language; we have no facilities to interpret Paddle breadth-first or in some other nonoperational manner. To illustrate the significance of this decision, consider the problem of choosing two different data structures in different parts of the program. An implementor may interactively decide which choices to make "in any order," using whatever strategy he feels is appropriate (breadth-first examination of alternatives, for example). The system, when it is **applying** the development to the program (for example, during a replay), will

---

[3] Also used in Hewitt's (full) Planner.

completely elaborate one of the choices and its dependencies even before introducing the other choice.

This distinction involves the differences between the development **process** and the development **structure**, to be described presently.

4. THE DEVELOPMENT PROCESS

Although the development structure is applied like a program to a specification to yield an implementation, the process of designing the implementation and its explanation is by no means so stylized. In general, the following scenario captures the normal activity of the implementor:

Repeatedly:

- Focus on a program fragment;
- Find an appropriate implementation strategy;
- Get the program into "condition" to allow application of the strategy;
- Apply the strategy;
- Simplify the resulting program.

Notice that the process is potentially recursive in two ways: conditioning the program may require that further subgoals in the process be met, and applying the strategy itself may require modification of several pieces of the program (as in the divide-and-conquer example above). This recursive structure must eventually be reflected in the development. It can be incorporated wholly beforehand (*a priori*) or afterward (*a posteriori*) as the explanation of that development.

An implementor normally switches back and forth between these two modes during any single session.

4.1. *A Priori* Explanation

A priori explanation corresponds to planning, or using an existing implementation strategy. This is certainly a frequent initial implementation approach, for high-level specifications are usually so intrinsically inefficient that previous experience with similar problems suggests an overall implementation design. For example, while text-processing systems are best specified as multiple pass algorithms, most programmers will implement such systems as single pass algorithms. Hence, most implementors will choose multiple pass merging as their topmost strategy.

To produce an *a priori* explanation using our system, the implementor must indicate the focus of attention on the program in the development, as well as the actual implementation plan. He generally creates a piece of development structure to express both the implementation plan and the focus of attention. He then applies the development structure to the specification.

When using *a priori* explanation, and therefore, applying a development to a specification, the implementor needs feedback as to exactly what is happening to the specification, in case his expectations are not met. Hence, *in our system*, the application of the development structure is **traced**. This gives the implementor exactly the same feedback as he would have had if he had done the transformations *a posteriori*.

Normally, something goes wrong during *a priori* development. Either the development plan contains undefined steps or a transformation's pattern or enabling conditions fail to match. When this happens the implementor becomes problem-driven rather than strategy-driven: he will produce an *a posteriori* explanation.

4.2. *A Posteriori* Explanation

When the implementor is not sure what transformation to apply next, or what portion of the program to focus on, or when problems arise with a planned development, he will switch his attention to the program itself. He may change the program, using editing commands and transformations. Often, such commands are used to condition the program for the transformation that was being attempted. When this happens, he may want the editing steps to be "bundled up" and inserted into the development structure, or he may want to make a new transformation which generalizes his editing steps and insert a call to it in the development. Support for both of these processes is provided in our system.

We emphasize that ultimately, it is the entirely automatic application of a development to a program to produce the resulting implementation that gives credence--and self-confidence--to the optimizer. Despite excursions into *a posteriori* explanation, the final implementation must appear to subsequent maintainers to have been produced entirely *a priori*.

5. REPLAY OF DEVELOPMENTS

Of course, the reason for having the development structure as a formal object in the first place is so that replaying the development (in part) on changed specifications is the normal mode of operation. Unfortunately, simply having the explanation for the implementation does not guarantee the ability to replay developments accurately. There are two basic problems:

- Replaying the development and getting errors when it was expected to work.
- Replaying the development and getting no error when the replay should not work.

Naturally, the latter problem is the most insidious, for the implementor will not know that the new development is flawed. These can arise from insufficient identification of assumptions in the original development or implicit assumptions in the system.

5.1. Unexpected Errors

We have no real-world experience with replay, since we have not "maintained" (i.e., changed) any of the example specifications yet. Nevertheless, a fair amount of it occurs even in a normal design: midstream in the design, one often decides to try the whole thing "from scratch," as though the entire

development were designed *a priori*, in order to test the accuracy of our development structure. From this experience, it is clear that the problems related to the development failing when we thought it would work are often problems of **focus**. The language we use is inadequate for expressing exactly which portion of the specification or development is being transformed. Generally, the language is simply too low level--it does not identify the pieces being transformed by using labelled program segments or high-level descriptions, like, for example, "the loop over characters." High-level editing notions as suggested by [Waters 82] must be incorporated to avoid this problem.

5.2. Unreported Errors

We have begun to use conventions to forestall problems of the second type above. First, we often express a "map" or "template" of what we believe the implementation looks like at different, key stages in the development.

Second, we have started identifying key stages in the development structure where a dynamic snapshot of the implementation should be presented to the implementor. In particular, although the tracing facility is extremely useful during the design of the development, it is just like any other tracing facility when the traced object becomes large: it is overwhelming. Hence, looking back to it for information during reimplementation would be time consuming.

We have found it quite useful to identify **major steps** in the development and print out the entire implementation state before and after those steps. Subsequent maintenance versions can be compared with the original major steps to decide on new strategies. Basically, this is one mechanism which allows the maintainer to check that his newly created development is "on track" with the old one when he intends for it to be.

Of course, the major issue of checking that (implicit) assumptions match is most difficult. Recent work on semantic matching by [Chiu 81] has solved part of the problem; systems can automatically compare two implementations and present semantic explanations of their differences to the user. However, this area remains completely open for solutions.

6. THE UNDERLYING PROGRAM MANIPULATION SYSTEM: POPART

POPART[4] [Wile 82] is a system developed in Interlisp [Teitelman 78] to provide the basis for a programming environment for **arbitrary** programming languages--in fact, for arbitrary languages describable in BNF. The tools provided for objects described by BNF grammars[5] include a parser, an editor, a pretty printer, a lexical analyzer, and a language-independent pattern-matching and replacement mechanism. In fact, the transformation system itself is one of these language-independent tools! A "pure" parser was produced initially as a reaction to systems that embed semantic processing in the syntactic parsing mechanism [Griss 76]--LISP itself seemed to be a preferable medium for expressing the semantics of parsed sentences. In fact, to support the set of tools mentioned, an abstract representation of all the information in the source language must be

---

[4] Producer of Parsers and Related Tools.

[5] Of course, a variant allowing regular expressions.

maintained--i.e., a "pure" parser **must** be used for such systems. The idea to provide tools for manipulating expressions in these languages arose from proposals by Balzer [Balzer 69, Balzer 73] and Yonke's Ph.D. dissertation establishing its feasibility [Yonke 75]. POPART is certainly related to recent efforts on programming language environments, such as Gandalf [Habermann 80, Feiler 80] and the environments for PL/CS [Teitelbaum 81] and Pascal [Kahn 75]. It also defines a language for program manipulation, and is thus related to the recent work of Cameron and Ito on grammar-based metaprogramming systems [Cameron 82].

A BNF grammar is used to generate an **abstract syntax** for the language; expressions are subsequently parsed by POPART into this abstract syntax. Thereafter, no other representation of the program exists--i.e., no stream of lexemes or characters. All tools work with the abstract syntax, variously converting strings into it and it into strings when communication with the user is necessary: the user always views and enters **source language**--he never sees the abstract syntax representation itself. This is quite different from the Gandalf system, but is consonant with Kahn's Pascal system, Mentor. POPART is embedded in the Interlisp interactive environment: it is a set of "commands" invoked just like any other Interlisp commands (EVALQUOTE). Hence, we should think of POPART not as a system, but as a set of augmentations to the already extensive Interlisp environment, provided to deal uniformly with objects described in BNF grammars.

POPART itself is intended to be a set of tools from which a **system designer** constructs and customizes a system. The default mechanisms provided to the designer support an environment in which a single object is always being edited (for each grammar known to POPART). The **user** of the editor has commands for moving about in the abstract representation of the object; he may go in, out, forward, and backward in the structure. He also can change the object, but only in ways that maintain the grammatical integrity of the object. Appendix I contains a transcript demonstrating the use of the POPART editor.

It is not the intent of this report to describe the POPART system in detail. Those portions relevant to understanding the transformation system (component) will be dealt with as they are encountered.

7. STRUCTURES EXPRESSED IN THE PADDLE LANGUAGE

The single most powerful feature of the POPART/Paddle system is that since Paddle itself is described as a language with a formal syntax, Paddle developments themselves may be manipulated by the user using the POPART primitives! This is the nature of the synergism derived from using generic, tool-based systems rather than pat encapsulations isolating users from the environment system.

The fact that the Paddle language is independent of the programming language means that the development structure mechanism can be a POPART tool. As was mentioned above, POPART editing **commands** can be written using Paddle's program manipulation facilities. Introducing Paddle comes full circle: we use POPART on Paddle, and then use Paddle in POPART.[6]

The Paddle development language is used to describe four different structures to POPART: Global Commands, Simplifications, Conditioners, and the Development.

---

[6]Note we have not yet used POPART and Paddle to implement POPART and Paddle

7.1. Global Commands

The global commands are simply parameterized macros that can be used in any of these POPART structures and that may be explicitly invoked as editing instructions when editing the program itself. For example, if one wanted an abbreviated way to find a conditional statement, he might define the command:

```
command FindIf() =
  Find !ConditionalStatement
```

This innocuous definition represents much of the complexity of the Paddle/POPART marriage, so we will belabor it a bit. First, there are conceptually three different languages involved here:

- the language of the development system, Paddle;
- the *primitive commands* of the development system (chosen to be POPART's editing commands);
- the programming language that represents the *program being transformed*.

Font Conventions

Different font conventions have been adopted for each of the different languages to help the reader differentiate them, as follows:

Paddle
Development Language -- optimize body, comments, and so forth
Development Keywords -- **each**, **by**, **first** ...
Global Command Names -- MergeLoops, FindCall...

Popart
Primitive Command Names -- *Find*, *Top*, *ReplaceAll*...

Programming Language
Programming Language -- *text*, *character*, *vary3*...
Programming Language Keywords -- ***begin***, ***end***, ***procedure***...

Notice that the Paddle global command FindIf above is defined to be the POPART editor *Find* command of a pattern in the programming language: !*ConditionalStatement*. It is necessary for POPART to support switching between grammars for such expressions to be parsed. The expression !*ConditionalStatement* indicates that anything syntactically derivable from the grammar nonterminal "ConditionalStatement" should match. *ConditionalStatement* represents a pattern variable in the pattern language used for the *Find* and *Replace* commands.

What are normally considered to be transformations are also definable as commands. For example, to replace a conditional whose predicate is the constant ***true*** with its ***then*** clause one could write the following:

```
command ReplaceWithThen() =
  begin
    first of
      Match if true
                then !Statement;
      Match if true
                then !Statement
                else !Statement #
    end;
    Replace !Statement
  end
```

The POPART editing commands *Match* and *Replace* are the primitives of the Paddle development language. Notice that here in a simple transformation we have used the conditional goal satisfaction mechanism of the Paddle language--the **first of** command.[7] Either pattern may match (an *if* statement with or without an ***else*** clause). The *Match* command differs from the *Find* command in that it is an "anchored search" for the pattern. The statement matched will subsequently be replaced by the ***then*** part. This replacement will only occur if (some option within) the **first of** command succeeded. Otherwise, the **first of** command will fail.

Finally, plans or strategies as described above may be included among the global commands:

```
command DivideAndConquer(function,set) =
    begin
      split set={e1,e2,...} into subsets s1, s2, ...;
        by
          choose from
             partitioning into s1={e1} and s2={e2,e3,...};
             binary partitioning into s1={e1...ek/2} and
                                      s2={ek/2+1...ek};
             basis partition s0, s1, s2, s4, ... s2i,
                where each en is a linear combination of the s2j
          end
      compute a related function, f1, on the subsets;
      combine values of f1 on subsets via a new function, f2;
      note  You must insure that function applied to set =
              f2 applied to {f1(s1), f1(s2), ...}
    end
```

Notice, in this command, the only predefined command is the *note* command!

[7] Lack of an "option" in the pattern language forces the use of the first of command. We contemplate the future use of POPART's BNF to specify patterns, thus eliminating this difficulty.

7.2. Simplifications

Paddle is also used to describe **simplifications** to the editor. Each time a *Replace* command is called in the editor, the resulting expression is checked for various simplifications. Some of these are described by the grammar designer to POPART, such as automatic removal of extra parentheses when nested constructs replace expressions in which the nesting is unnecessary. In addition, a single Paddle **Simplification** command is always applied to the modified program when a replacement is made. For example, the ReplaceWithThen command defined above would be a reasonable simplification command to try. If we had an analogous command, ReplaceWithElse,

---

```
command ReplaceWithElse() =
  begin
    Match if false
                then !Statement
                else !Statement# ;
    Replace !Statement#
  end
```

---

we might include these in the simplification structure:

```
first of
  ReplaceWithThen;
  ReplaceWithElse
   *  *  *
end..
```

7.3. Conditioning

During the transformation process, it is frequently the case that a transformation's pattern will fail to match when the implementor thought it would (or should). He will then have to divert his attention from transforming to "getting the program into condition" to be transformed. Normally, this process of **conditioning**[8] the program will merely involve the application of a simple, equivalence-preserving transformation to the program.

POPART provides conditioning at the syntactic level within the *Find* and *Match* commands. The system builder builds tables which direct this activity by classifying productions as having associative, commutative, or nested fields. POPART will then automatically rewrite expressions using this information to condition it to match.

Conditioning is also provided for in the Paddle language in a manner analogous to simplification: A conditioning command is applied to the current expression to attempt to change it so that it will match

---

[8] We previously called this "jittering," but find the connotation distasteful.

a pattern that has failed to match.[9] For efficiency reasons, this will require preprocessing of the conditioning commands, to see if the pattern being matched could be produced by a *Replace* command in the conditioning command. For example, if the following conditioning commands were given to the system,

```
begin
  command IntroduceThen() =
    begin
      Match !Statement;
      Replace if true
                    then !Statement
    end;
  command IntroduceElse() =
    begin
      Match !Statement;
      Replace if false
                    then null
                    else !Statement
    end
end
```

and the user attempted

```
Match if !Predicate then !ActionInvocation
```

when the current expression was

```
TextRemove[text,character]
```

the conditioner would have to notice that the IntroduceThen command produces a conditional statement with the same format as the pattern being matched (the argument to the *Match*). It would then attempt to execute the command. If it succeeded, and the resulting expression matches, it is done.

```
if true then TextRemove[text,character]
```

Otherwise, it has a choice: it can either attempt to make the command succeed or try other *conditioning commands*. We will probably implement this mechanism as a breadth-first search with a very early cutoff (depth 2). This mechanism is significant because Paddle is used to express all program manipulations and because much of the conditionality currently embedded in plans and

[9] This is not implemented yet.

developments to handle local variability can be factored out and put into the conditioning mechanism. This will greatly simplify and clarify the plans and developments while insuring that this conditioning capability is uniformly applied.

7.4. The Development Process: A Simple Example

Of course, the development structure itself is the major focus of attention here.

Appendix II is an actual transcript of a development of an implementation for the toy specification designed in Appendix I. The two transcripts together--Appendix I and Appendix II--have been constructed to be "self-explanatory"; many details of the POPART/Paddle system can be gleaned from careful reading of them.

The development process described in the Appendix typifies the nature of interactive program and development manipulation. Two characteristics stand out:

- The development **process** is much more verbose and tedious than the final development explanation.
- The development **process** is quite error-prone.

Both argue strongly that a **transcript** of the development process is inappropriate documentation of the optimization itself.

7.5. Text Compression: An Extended Example Development

The actual development structure arrived at in the above example was too trivial to actually demonstrate most of the interesting issues involved in structuring explanations for later consumption. Hence, a related but considerably longer example development has been presented in its final form in Appendix III. This describes the partial implementation of the program:

```
begin
  action
    savet[text |list of character, pred |predicate]
    definition loop(any character) suchthat character in text
                        unless pred(character)
                        do removet[text, character];
  relation
    redundant←space(character, seq |list of character)
    definition successort(seq, *, character) isa space
    and character isa space;
  loop(any linefeed) suchthat linefeed in text
        do atomic insert linefeed isa space;
                  delete linefeed isa linefeed
           end atomic;
  savet[text, 'a character ||character isa alphanumeric or character isa
           space];
  loop(any space) suchthat space in text and redundant←space(space,
                                                              text)
        do removet[text, space]
end..
```

This example was first worked out (manually, without system aids) in [Balzer 76]. In that paper, approximately the same development strategy as we are now able to describe formally was suggested as the desirable way of accomplishing the optimization. Our formal representation of that strategy is now:[10]

```
begin
  Pretty;
  MajorStep substitute savet definition for call
    by Unfold savet;
  MajorStep obtain a single loop
    by !POTAndCommands;
  MajorStep optimize loop body
    by !POTSeqCommands;
  MajorStep pick data representations
end..
```

The primitive command MajorStep causes the program to be printed out after its refinement has been executed. As was mentioned above, the verbatim trace of the executed primitive commands is not very valuable after-the-fact documentation. It is much more informative for the development structure to dynamically identify key steps which subsequent optimizers should use as "checkpoints" that the maintenance they perform is "on track" with the previous optimization. Thus, Appendix IV is included as an important (though easily regenerated) adjunct to the formal development. It

[10] *The summarization of this development has been produced automatically using the POPART pretty-printer's level control mechanism. The references to !POTAndCommands and !POTSeqCommands have been inserted automatically; they are merely "stubs" whose values are printed subsequently in the transcript.*

represents the actual application of the development in Appendix III to the initial program. The tracing of the primitive commands has been turned off, yielding a much clearer picture of the development process itself.

8. PROBLEMS AND FUTURE RESEARCH

We believe we are in an excellent position to begin to do experimental research on development styles and the fundamental support necessary to make transformation systems realistic. The POPART and Paddle facilities are all implemented and function as described. Extensions to the system will arise from extensive experiments with large, realistic examples. I expect future experience to duplicate the past: Paddle commands are defined to approximate some facility that seems desirable. Experimentation with it leads to its inclusion as a primitive command or its rejection.

We are aware that these specific areas still need considerable attention:

8.1. A Separate Goal Structure

Some goals cannot actually be expressed as independent, even though there appear to be two separate tasks being accomplished. For example, in the divide-and-conquer plan above, f_1 and f_2 are neither independent nor sequentially dependent. This defect may require that a separate goal structure be maintained (a noninterpretable structure). This is actually necessary for any reasonable interpretation of codependent goals or even entirely independent goals: the operationality of the development structure is too constraining to express these concepts adequately.

8.2. Styles to Support Maintenance

Exactly what mechanisms--like checkpoint snapshots of the optimization in progress--are necessary to facilitate maintenance activities on the specifications? How should the optimizer describe the editor's focus of attention on the program so as to remain general enough so that simple changes do not cause the attention to "drift," and yet be specific enough that replays do not work with just any new specification?

Although we described the development structure as "an explanation" of the development, there are other explanatory styles of more utility. For example, [Swartout 81] uses a similar structure to produce answers to individual questions (about programs); the same might be used to justify development steps on a more localized basis.

8.3. Generic Transformations

The sequential composition, refinement subordination, and choice constructs provide the basis for creating packages that encapsulate a structured knowledge base of interrelated decisions. Their use results in selection of an implementation for some higher level goal (for example, "divide and conquer"). Packaging development strategies in ways that exhibit intelligent reaction to information provided by the user is an important issue for future research: how to describe or suggest the appropriateness of certain choices and to order dynamically the consideration of decisions.

8.4. Increased Automation

It is clear that automatic facilities are necessary for a useful system. Two major areas need work: predicate maintenance--flow analysis as well as domain dependent "predicate pushing," and automatic conditioning--including choosing appropriate transformations based on hints from the user.

8.5. Developments in Other Domains

We mentioned above that the set of primitive commands underlying Paddle need not be an editing language. We have two applications to quite different domains in which we wish to study the use of Paddle. First, we have already experimented with the use of Paddle in Affirm. Affirm [Gerhart 80] is a program verification/theorem proving system. Its command set has been used as a Paddle primitive node language. In that context, Paddle provides a mechanism for defining and invoking proof strategies. Paddle developments are applied to a state consisting of a set of theorems to be proved and a set of program specifications to be verified. The developments (may) represent entire program validations. A language-dependent version of some of these same Paddle development notions is captured in the proof metalanguage for LCF [Gordon 78].

Another application in which Paddle may be useful is for specification **design**. In particular, the design decisions used in arriving at an initial specification should be documented as thoroughly as those used to arrive at an implementation. With a primitive node language devoted to describing the goals achieved when features are introduced into specifications we expect Paddle to provide a suitable framework for such design documentation. This will not be like Caine and Gordon's PDL [Caine 75], but will instead document the design process; i.e., the final development structure will not contain the program pieces in the leaves, but rather will tell how the specification changes between design stages.

We must emphasize that the directions taken for the future work will be based principally on the necessities demanded by a large example. If predicate maintenance does not seem to be a significant bottleneck, we will ignore it to the benefit of other areas. We believe we have laid the groundwork for extensive experimentation into the appropriate facilities for realistic transformation systems of the future.

I. POPART EDITOR SAMPLE TRANSCRIPT

Font Conventions

Paddle
Development Language -- optimize body, comments, and so forth
Development Keywords -- **each**, **by**, **first** ...
Global Command Names -- MergeLoops, FindCall...

Popart
Primitive Command Names -- *Find*, *Top*, *ReplaceAll*...

Programming Language
Programming Language -- *text*, *character*, *vary3*...
Programming Language Keywords -- ***begin***, ***end***, ***procedure***...

```
1←note  The numbers on the left are "interaction numbers," each transaction
...     with Interlisp is recorded.  The command being executed right now
...     is simply a "comment" command.  The first thing a user/optimizer
...     does is focus POPART's attention on a grammar.

2←EditProgram

3←note  Then, the user Sets the attention of POPART to an expression or
...     statement in that grammar.  I'll enter a short specification:

4←Set
begin loop (any character) suchthat character in text do
          if character isa linefeed
            then TextReplace[text,character, a space];
     KeywordSearch[text,keys]
end..

5←note  The representation POPART maintains is a parsed version of the
...     specification I entered in a language called Gist.  This representa-

6←note  tion is simply a list structure which is of no concern to the
...     optimizer himself, for he can examine objects as though they were
...     in the source language by asking for the expression to be printed
...     "in a pretty fashion."

7←Pretty
begin
  loop(any character) suchthat character in text
         do if character isa linefeed
                  then TextReplace[text, character,
                                   a space];
  KeywordSearch[text, keys]
end..
8←note  In fact, POPART even puts out font information when it is instructed
...     to as it was here.  As with any structure editor, one moves about
...     with reference to the structure, rather than with reference to lines
...     or characters.  We can move into the current expression using the
```

```
...     "In" command:

9←In

10←Pretty
loop(any character) suchthat character in text
        do if character isa linefeed
                then TextReplace[text, character, a space]..
11←note  We can move to the next statement by using the "Next" command:

12←Next

13←Pretty
KeywordSearch[text, keys]..
14←note  This little specification is supposed to "compress" text so that a
...      keyword search can be made for any of a set of keys.  I forgot to
...      put in a statement to remove nonalphanumeric characters.  I can
...      do this using the "Before" command, which inserts its argument
...      before the current statement.

15←Before
loop ((any character) suchthat character in text) unless
   alphanumeric(character) or character isa linefeed
  do TextRemove[text,character]..

16←note  I can move to the outermost expression by using the "Top" command:

17←Top

18←Pretty
begin
  loop(any character) suchthat character in text
          do if character isa linefeed
                  then TextReplace[text, character,
                                   a space];
  loop((any character) suchthat character in text)
          unless alphanumeric(character) or character isa linefeed
          do TextRemove[text, character];
  KeywordSearch[text, keys]
end..
19←note The correction I made is reflected in the current expression.
...     Oops.  I did not mean to ask if character isa linefeed, but rather
...     if character isa space in that insertion I just made.  I can get
...     back to where I was in several ways.  One is to "UNDO" the command
...     which got me to the Top, 17.  Another is to use the pattern matcher
...     to Find the appropriate test.

20←Find
character isa linefeed..
Not unique.

21←note  Since there is more than one occurrence in the program, the
...      matcher has warned me that I may not have found the one I want.
...      I can print the context of the match using a command:
```

```
22←PrintContext
if character isa linefeed
     then TextReplace[text, character, a space]..
23←note  That is not the one I want.  I can find the next occurrence of the
...      last pattern matched by using the "Refind" command:

24←Refind
Last match.

25←note  Clearly, this has to be it.  We can replace the current expression
...      using the Replace command:

26←Replace
character isa space..

27←Top

28←note  This is the version of the program I wanted.  Let's save it in
...      case we break something:

29←Snapshot mini-eg

30←Pretty
begin
  loop(any character) suchthat character in text
         do if character isa linefeed
                  then TextReplace[text, character,
                                   a space];
  loop((any character) suchthat character in text)
         unless alphanumeric(character) or character isa space
         do TextRemove[text, character];
  KeywordSearch[text, keys]
end..
31←note  To demonstrate some of the power of the pattern matcher, consider
...      how we would find the loop generator with an "unless" clause in it:

32←Find
!GeneratorSecondary unless !Predicate..
Unique.

33←note  The user must know the grammar nonterminals--GeneratorSecondary
...      and Predicate--in order to write this pattern.  The values of the
...      pattern variables with the same names can be seen using the Value
...      command:

34←Value GeneratorSecondary
((any character) suchthat character in text)..
35←Value Predicate
alphanumeric(character) or character isa space..
36←Pretty
((any character) suchthat character in text)
  unless alphanumeric(character) or character isa space..
37←note  We can even make the system rewrite the current expression to match
...      a pattern--a concept we call "conditioning."  For
...      example, if we want to match two loops in a row, we can write:
```

```
38←Top

39←Find
begin !LoopingStatement; !LoopingStatement# end..
Unique.

40←note  To demonstrate that the specification has been rewritten, we can
...      look at the current expression as well as the outermost expression:

41←Pretty
begin
  loop(any character) suchthat character in text
        do if character isa linefeed
                then TextReplace[text, character,
                                        a space];
  loop((any character) suchthat character in text)
        unless alphanumeric(character) or character isa space
        do TextRemove[text, character]
end..
42←Top

43←Pretty
begin
  begin
    loop(any character) suchthat character in text
          do if character isa linefeed
                  then TextReplace[text, character,
                                          a space];
    loop((any character) suchthat character in text)
          unless alphanumeric(character) or character isa space
          do TextRemove[text, character]
  end;
  KeywordSearch[text, keys]
end..
44←note  As before, the metavariables have been set:

45←Value LoopingStatement
loop(any character) suchthat character in text
      do if character isa linefeed
              then TextReplace[text, character, a space]..
46←Value LoopingStatement#
loop((any character) suchthat character in text)
      unless alphanumeric(character) or character isa space
      do TextRemove[text, character]..
47←note  The program is in a rather "uncanonical" form.  Normally, if the
...      user enters extraneous begin end pairs, the system will
...      automatically simplify them away.

48←In

49←In

50←Pretty
loop(any character) suchthat character in text
```

```
        do if character isa linefeed
                then TextReplace[text, character, a space]..
51←Replace
begin SillyCall[text] end..

52←Top

53←Pretty
begin
  begin
    SillyCall[text];
    loop((any character) suchthat character in text)
          unless alphanumeric(character) or character isa space
          do TextRemove[text, character]
  end;
  KeywordSearch[text, keys]
end..
54←note  Notice that the SillyCall was extracted from the extraneous begin
...      end pair.
```

II. SAMPLE DEVELOPMENT TRANSCRIPT

```
1←note  This transcript will demonstrate how the development and its
...     related structures are constructed and applied to the specification.
...     We must first start off with the specification entered previously
...     in Appendix I.
...

2←EditProgram

3←Unsnapshot mini-eg
FILE CREATED  1-Apr-81 16:33:57
((VARS POECurrentObject POEGlobalBindings) (P (POEPrintBindingCommand)))

4←Pretty
begin
  loop(any character) suchthat character in text
        do if character isa linefeed
                then TextReplace[text, character,
                                      a space];
  loop((any character) suchthat character in text)
        unless alphanumeric(character) or character isa space
        do TextRemove[text, character];
  KeywordSearch[text, keys]
end..
5←note  Normally, when we wish to develop an optimization plan, we simply
...     edit the development object:

6←EditDevelopment

7←note  Imagine that we would like to merge the first two loops in the
...     specification.  We can write a development which does this in
...     many different ways.  We could write the steps which transform
```

```
...     the program in-line, or we could invoke a transformation which
...     is globally defined.  Let's do the latter by using the Set
...     command--the same as we used for the specification above--to
...     an expression in the development language, Paddle.

8←Set
MergeLoops..

9←note  This parses as a command which it knows nothing about.  Let's
...     define the command as a global command and then try to apply it
...     to the program.  We must first switch contexts and edit the
...     global command structure:

10←EditGlobalCommands

11←note  Again, we use the Set command to define a Paddle expression for
...      the command which merges loops.  The command itself must look
...      for two loops in sequence with the same generator part, and then
...      it must replace the two with a single loop with bodies in
...      sequence.  Normally, we need enabling conditions to guarantee
...      the bodies do not interfere, etc., but for the purposes of
...      demonstration let's ignore that presently:

12←Set
begin command MergeLoops =
        begin Find begin loop !SetExpression do !Statement;
                          loop !SetExpression do !Statement#
                    end;
              Replace loop !SetExpression
                        do begin !Statement; !Statement# end
        end
end..

13←note  On type-in, I have tried to indent fairly carefully to indicate the
...      way the different begin end pairs--from Paddle and Gist--match up.
...      The pretty version from the system should be more readable:

14←Pretty
begin
  command MergeLoops() =
  begin
    Find begin
           loop !SetExpression
                  do !Statement;
           loop !SetExpression
                  do !Statement#
         end;
    Replace loop !SetExpression
                    do begin
                         !Statement;
                         !Statement#
                       end
  end
end..
15←note  Ok.  Now we have a transformation defined, we have a development
```

```
...       structure and we have a specification which we may apply it to.
...       Let's try it and see what happens.

16←EditDevelopment

17←note  We do this by using the following command:

18←ApplyToProgram
19←Find begin
       loop !SetExpression
              do !Statement;
       loop !SetExpression
              do !Statement#
     end..
No match.
Fix development tree.

20←note  Line 19 was inserted into the transcript as a result of attempting
...       to apply the development to the program.  In fact, the development
...       structure has been expanded to include the definition of MergeLoops
...       and we are left editing the Find command which does not match, viz.

21←Pretty
Find begin
       loop !SetExpression
              do !Statement;
       loop !SetExpression
              do !Statement#
     end..
22←Top

23←Pretty
MergeLoops
  by begin
       Find begin
              loop !SetExpression
                     do !Statement;
              loop !SetExpression
                     do !Statement#
            end;
       Replace loop !SetExpression
                     do begin
                          !Statement;
                          !Statement#
                        end
     end..
24←note  Let's go over to the program and see what is wrong--why doesn't
...       the pattern match?

25←EditProgram

26←Pretty
begin
  loop(any character) suchthat character in text
         do if character isa linefeed
```

```
                then TextReplace[text, character,
                                 a space];
  loop((any character) suchthat character in text)
        unless alphanumeric(character) or character isa space
        do TextRemove[text, character];
  KeywordSearch[text, keys]
end..
27←note  Aha.  The second loop has an "unless" clause which is causing the
...      set expressions to be different on the two loops.  In the context
...      of a loop, we can move the unless clause into a conditional in the

28←note  loop body.  We can either do that now to the program or go over
...      to the development and describe how to do it there.  Let's do the
...      latter.

29←EditDevelopment

30←UNDO Top
Top undone.
31←Pretty
Find begin
       loop !SetExpression
            do !Statement;
       loop !SetExpression
            do !Statement#
     end..
32←note  We are now back on the Find command which would not match.  We can
...      refine this command to include explicit "conditioning" to the
...      specification which will cause the LoopMerging pattern to match.

33←Replace
$$ by begin Find loop !GeneratorSecondary unless !Predicate do !Statement;
            Replace loop !GeneratorSecondary do if ~(!Predicate) then
                        !Statement
      end..

34←note  $$ always refers to the current expression in the editor.  Hence,
...      the last statement merely added the unless conditioning step to
...      the Find command.

35←Pretty
Find begin
       loop !SetExpression
            do !Statement;
       loop !SetExpression
            do !Statement#
     end
  by begin
       Find loop !GeneratorSecondary
                 unless !Predicate
                 do !Statement;
       Replace loop !GeneratorSecondary
                    do if~( !Predicate)
                            then !Statement
     end..
```

```
36←note  Now we can attempt to continue the development from this point
...      by again asking the system to Apply the current expression to
...      the specification, from the point at which the editor is focused.

37←ApplyToProgram
38←Find loop !GeneratorSecondary
           unless !Predicate
           do !Statement..
Unique.

39←Replace loop !GeneratorSecondary
              do if~( !Predicate)
                     then !Statement..

40←Find begin
       loop !SetExpression
              do !Statement;
       loop !SetExpression
              do !Statement#
     end..
No match.
Fix development tree.

41←note  Aha.  After finding the second loop and fixing the generator
...      to match the first, the editor's focus in the specification
...      is on the loop.  Hence, the loop merging pattern did not match.
...      We must fix this by refocusing after the conditioning.

42←note  Let's look at the program first.

43←EditProgram

44←Pretty
loop(any character) suchthat character in text
       do if~(alphanumeric(character) or character isa space)
              then TextRemove[text, character]..
45←Top

46←EditDevelopment


47←note  Now we can apply the development from the point in error and
...      continue.  N.B.:  This development no longer reflects the exact
...      optimization history, because of the Top command in line 45
...      that is no longer accounted for.  We will return to this.

48←ApplyToProgram
49←Find begin
       loop !SetExpression
              do !Statement;
       loop !SetExpression
              do !Statement#
     end..
Unique.
```

```
50←Replace loop !SetExpression
              do begin
                   !Statement;
                   !Statement#
                 end..


51←note  Now we are done.  We are on the specification side.  The "final"
...      program looks like:

52←Top

53←Pretty
begin
  loop(any character) suchthat character in text
         do begin
              if character isa linefeed
                   then TextReplace[text, character,
                                          a space];
              if~(alphanumeric(character) or character isa space)
                   then TextRemove[text, character]
            end;
  KeywordSearch[text, keys]
end..
54←note  Now we should attend to making the development match the actual
...      optimization, manipulating the development structure using the
...      editor.

55←EditDevelopment

56←Top

57←Pretty
MergeLoops
  by begin
       Find begin
              loop !SetExpression
                     do !Statement;
              loop !SetExpression
                     do !Statement#
            end
         by begin
              Find loop !GeneratorSecondary
                           unless !Predicate
                           do !Statement;
              Replace loop !GeneratorSecondary
                              do if~( !Predicate)
                                       then !Statement
            end;
       Replace loop !SetExpression
                      do begin
                           !Statement;
                           !Statement#
                         end
     end..
```

```
58←note  First we will make a transformation out of the conditioning step,

59←note  and then we can put the Top invocation near it.

60←Field
Allowed fields are: POTCommand and POTPrimitiveCommand.

61←note  The POTCommand field is the refinement part of the command.

62←Field POTCommand

63←NAME refinement 62
refinement
64←note  This is an Interlisp command which now makes refinement mean the
...      same as line 62.

65←In

66←refinement

67←Pretty
begin
  Find loop !GeneratorSecondary
               unless !Predicate
               do !Statement;
  Replace loop !GeneratorSecondary
                  do if~( !Predicate)
                          then !Statement
end..
68←note  We can now set a transformation language, Paddle, metavariable
...      to this command in order to move its definition over to the
...      global commands area.

69←SetGlobal
POTCommand = $$..

70←Value POTCommand
begin
  Find loop !GeneratorSecondary
               unless !Predicate
               do !Statement;
  Replace loop !GeneratorSecondary
                  do if~( !Predicate)
                          then !Statement
end..
71←note  This conditioning step can be replaced with a call to the
...      appropriate new command definition followed by a call to Top.

72←Replace
begin UnlessDefinition; Top end..

73←note  And now we can go and define the global command UnlessDefinition.

74←EditGlobalCommands
```

```
75←In

76←After
command UnlessDefinition = !POTCommand..

77←Top

78←note  This leaves the commands:

79←Pretty
begin
  command MergeLoops() =
  begin
    Find begin
           loop !SetExpression
                  do !Statement;
           loop !SetExpression
                  do !Statement#
         end;
    Replace loop !SetExpression
                   do begin
                        !Statement;
                        !Statement#
                      end
  end;
  command UnlessDefinition() =
  begin
    Find loop !GeneratorSecondary
                 unless !Predicate
                 do !Statement;
    Replace loop !GeneratorSecondary
                   do if~( !Predicate)
                            then !Statement
  end
end..
80←EditDevelopment

81←Top

82←note  With the final development:

83←Pretty
MergeLoops
  by begin
       Find begin
              loop !SetExpression
                     do !Statement;
              loop !SetExpression
                     do !Statement#
            end
         by begin
              UnlessDefinition;
              Top
            end;
       Replace loop !SetExpression
```

```
                    do begin
                         !Statement;
                         !Statement#
                       end
     end..
84←note  We can be sure this is accurate by applying it to the original
...      program.

85←EditProgram

86←Unsnapshot mini-eg
FILE CREATED  1-Apr-81 16:33:57
((VARS POECurrentObject POEGlobalBindings) (P (POEPrintBindingCommand)))

87←EditDevelopment

88←ApplyToProgram
89←Find loop !GeneratorSecondary
           unless !Predicate
           do !Statement..
Unique.

90←Replace loop !GeneratorSecondary
              do if~( !Predicate)
                      then !Statement..

91←Top..

92←Find begin
       loop !SetExpression
             do !Statement;
       loop !SetExpression
             do !Statement#
     end..
Unique.

93←Replace loop !SetExpression
              do begin
                   !Statement;
                   !Statement#
                 end..


94←Pretty
loop(any character) suchthat character in text
       do begin
            if character isa linefeed
                 then TextReplace[text, character,
                                        a space];
            if~(alphanumeric(character) or character isa space)
                 then TextRemove[text, character]
          end..
95←note  Indeed.
```

III. THE DEVELOPMENT OF A TEXT COMPRESSOR

The development which follows is intended to be applied to a text compression specification shown in Appendix IV. The global commands to which it refers are given in Appendix V, and the implicit simplifications performed on the specification are in Appendix VI.

Several commands used in this development were not explained in the text; many involve the distinction between global and local pattern variables. If a *Match* or *Find* command contains pattern variables in its pattern, the variables are local. Their definitions (names and values) are flushed on the next *Match* or *Find*. If it is desired to keep the value of a pattern variable across subsequent applications of the *Match* or *Find*, the variable must be made global. This is done using the *SetGlobal* command. The variable must later be removed using the *RemoveGlobal* command (we intend to replace this notion with a scope model in the future--the Paddle syntax has it built in (see Appendix VII)--but the system cannot yet deal with these variables).

A frequently used primitive command is the *Map* command. This is like a *Match* command, but it makes all the pattern variables into global pattern variables (flushing previous definitions if there were any). This is useful for describing to subsequent maintainers the exact format the optimizer believes the program to have at any given moment.

```
88←vp* 4
begin
  Pretty;
  MajorStep substitute savet definition for call
    by Unfold savet;
  MajorStep obtain a single loop
    by !POTAndCommands;
  MajorStep optimize loop body
    by !POTSeqCommands;
  MajorStep pick data representations
end..

where

!POTAndCommands=
        each merge 1st two loops
               by !POTSeqCommands#;
             merge in remaining loop
               by !POTSeqCommands##
        end..

!POTSeqCommands=
        begin
          command LoopBody() =
          begin
            Top;
            In;
            Last;
            Field Statement
          end;
          LoopBody;
```

```
        note We are positioned at the loop body;
        ReplaceAll characteristic( !ObjectExpression)
                                ==>character in text;
        Map case character of
                linefeed=> !Statement#;
                alphanumeric=>comment NOOP
                                end comment;
                space=> !Statement# # #;
                othercase=> !Statement# #
            end case
      end..

!POTSeqCommands#=
      begin
        Top;
        Map begin
              !DeclarationStatement;
              !LoopingStatement;
              !LoopingStatement#;
              !LoopingStatement# #
            end;
        begin
          note Condition the program to match generator expressions,
              first of putting in a supertype generator of linefeed and then
              removing the unless clause on the second loop;
          Find !LoopingStatement;
          SupertypeGenerator character;
          Find !LoopingStatement#;
          UnlessDefinition;
          Top
        end;
        MergeLoops;
        MajorStep First2LoopsMerged
      end..

!POTSeqCommands##=
      begin
        begin
          note Ideally we would probably like to say something like:
              MergeLoops using SetToSequence;
          Top;
          note Now we want to change the generators to sequential
              generators, because we know that the test for redundant
              spaces requires a sequential scan across the input to insure
              loop bodies are noninterfering. ;

          Map begin
                !DeclarationStatement;
                !LoopingStatement;
                !LoopingStatement#
              end;
          each !POTSeqCommands;
               !POTSeqCommands###
          end;
          Top
```

```
        end;
        MergeLoops
      end..

!POTSeqCommands=
      begin
        Find !LoopingStatement;
        SetToSequence
      end..

!POTSeqCommands###=
      begin
        Find !LoopingStatement#;
        Field SetOrSequenceExpression;
        GeneratorAndWhen:
        Out;
        WhenDefinition;
        SupertypeGenerator character;
        SetToSequence
      end..
```

IV. THE APPLICATION OF THE DEVELOPMENT TO THE SPECIFICATION: "REPLAY"

The development structure of Appendix III was applied to the (first) specification below to produce the trace below. The final program is the program after a single loop body was obtained. The actual transcript quits when the *Map* as a *case* statement was found to be impossible (at which point the user is left editing the development structure positioned on the failing *Map* command.)

```
69←ApplyToProgram
begin
  action
    savet[text |list of character, pred |predicate]
    definition loop(any character) suchthat character in text
                        unless pred(character)
                        do removet[text, character];
  relation
    redundant←space(character, seq |list of character)
    definition successort(seq, *, character) isa space
    and character isa space;
  loop(any linefeed) suchthat linefeed in text
        do atomic insert linefeed isa space;
                  delete linefeed isa linefeed
           end atomic;
  savet[text, 'a character ||character isa alphanumeric or character isa
           space];
  loop(any space) suchthat space in text and redundant←space(space,
                                                              text)
        do removet[text, space]
end..
```

MANUAL UNFOLDING STEP WENT HERE

```
Major Step: substitute savet definition for call
begin
  relation
    redundant←space(character, seq |list of character)
    definition successort(seq, *, character) isa space
    and character isa space;
  loop(any linefeed) suchthat linefeed in text
        do atomic insert linefeed isa space;
                  delete linefeed isa linefeed
           end atomic;
  loop((any character) suchthat character in text)
        unless character isa alphanumeric or character isa space
        do removet[text, character];
  loop(any space) suchthat space in text and redundant←space(space,
                                                              text)
        do removet[text, space]
end..


Major Step: First2LoopsMerged
begin
  relation
    redundant←space(character, seq |list of character)
    definition successort(seq, *, character) isa space
    and character isa space;
  loop(any character) suchthat character in text
        do begin
             if character isa linefeed
                  then atomic insert character isa space;
                              delete character isa character
                       end atomic;
             if characteristic((any character) suchthat character in
                                                        text)
                  then if not(character isa alphanumeric or character
                                                   isa space)
                            then removet[text, character]
           end;
  loop(any space) suchthat space in text and redundant←space(space,
                                                              text)
        do removet[text, space]
end..


Major Step: obtain a single loop
begin
  relation
    redundant←space(character, seq |list of character)
    definition successort(seq, *, character) isa space
    and character isa space;
  loop(any character) suchthat character in text
        do begin
             if character isa linefeed
```

```
                then atomic insert character isa space;
                            delete character isa character
                     end atomic;
           if characteristic((any character) suchthat character in
                                    text)
                then if not(character isa alphanumeric or character
                                    isa space)
                           then removet[text, character];
           if characteristic(text named character)
                then if character isa space
                           then if redundant+space(character,
                                                    text)
                                     then removet[text,
                                                    character]
      end
end..
```

HERE THE MAP ONTO CASE COMMAND FAILED

V. GLOBAL COMMAND DEFINITIONS

These "commands" form a library from which the optimizer chooses his optimization strategies, plans, and transformations.

```
command GeneratorAndWhen() =
first of
  begin
    Match while there exists !QuantifierRole || !LogicalSecondary
      and !LogicalFactor;
    Replace while there exists !QuantifierRole || !LogicalSecondary
        suchthat !LogicalFactor
  end;
  begin
    Match(any !Role) suchthat !LogicalSecondary
      and !LogicalFactor;
    Replace((any !Role) suchthat !LogicalSecondary) suchthat !
        LogicalFactor
  end
end;
command WhenDefinition() =
begin
  Match loop !GeneratorSecondary suchthat !Predicate
              do !Statement;
  Replace loop !GeneratorSecondary
               do if !Predicate
                        then !Statement
end;
command UnlessDefinition() =
begin
  Match loop !GeneratorSecondary
              unless !Predicate
              do !Statement;
```

```
    Replace loop !GeneratorSecondary
                    do if not( !Predicate)
                            then !Statement
  end;
  command SupertypeGenerator(SupType) =
  begin
    first of
      Find loop while there exists !QuantifierRole ||( !Variable in !
                                                                SetTerm)
                    do !Statement;
      Find loop(any !Role) suchthat !Variable in !SetTerm
                    do !Statement
    end;
    SetGlobal SetTerm;
    SetGlobal Variable;
    Replace !Statement;
    ReplaceAll !Variable==>SupType;
    Replace loop(any SupType) suchthat SupType in !SetTerm
                    do if SupType isa !Variable
                            then$$;
    RemoveGlobal SetTerm;
    RemoveGlobal Variable
  end;
  command SetToSequence() =
  begin
    Find loop(any !Role) suchthat !Variable in !GeneratorSecondary
                  do !Statement;
    Replace loop !GeneratorSecondary named !Variable
                    do !Statement
  end;
  command MergeLoops() =
  begin
    Find begin
            loop !SetExpression
                    do !Statement;
            loop !SetExpression
                    do !Statement#
          end;
    Replace loop !SetExpression
                    do begin
                            !Statement;
                            if characteristic( !SetExpression)
                                    then !Statement#
                          end
  end
end..
```

VI. SIMPLIFICATIONS

These commands are applied to each replacement on the program side until the overall command fails. This accomplishes localized canonicalization of the program at all times.

```
first of
  begin
    Find if true
              then !Statement;
    Replace !Statement
  end;
  begin
    Find if true
              then !Statement
              else !Statement# ;
    Replace !Statement
  end;
  begin
    Find if !Predicate# # #
              then !Statement# # #
              else !Statement# # # ;
    Replace !Statement# # #
  end;
  begin
    Find if false
              then !Statement;
    first of
          Delete;
          Replace comment NOOP
                    end comment
    end
  end;
  begin
    Find if false
              then !Statement
              else !Statement# ;
    Replace !Statement#
  end;
  begin
    Find false and !LogicalFactor;
    Replace false
  end;
  begin
    Find true and !LogicalFactor;
    Replace !LogicalFactor
  end;
  begin
    Find !LogicalSecondary and false;
    Replace false
  end;
  begin
    Find !LogicalSecondary and true;
    Replace !LogicalSecondary
  end;
  begin
    Find false or !LogicalTerm;
    Replace !LogicalTerm
  end;
  begin
    Find !LogicalFactor or false;
```

```
    Replace !LogicalFactor
  end;
  begin
    Find true or !LogicalTerm;
    Replace true
  end;
  begin
    Find !LogicalFactor or true;
    Replace true
  end;
  begin
    Find not true;
    Replace false
  end;
  begin
    Find not false;
    Replace true
  end
end..
```

VII. DEVELOPMENT LANGUAGE GRAMMAR

Popart Grammar Conventions

```
  :=  Definition e.g., A := B means define A to be a B
   |  Alternation e.g., A|B matches an A or a B
   '  Terminal Symbol e.g., '+|'- matches the constants + or -
      Concatenation e.g., AB matches an A followed by a B
  ()  Pattern Expression e.g., ('+|'-)A matches an A preceded by
                                    either a + or a -
   +  Repetition e.g., A+ matches any number (>0) of instances of As
   ↑  Lists e.g., A↑'; matches one or more As separated by ;s
LEXEME  Arbitrary Terminal Symbol, used for identifiers, numbers, etc.,
                                    almost always explicitly filtered.
  |>  Filter, LISP function following filters is applied after the pattern
              matches to further restrict the parse.
  ||  Compaction: abstract syntax tree for this production is represented
                  more compactly than normal.
Name, Name# Nonterminal Production: # sign used to distinguish multiple
                  occurrences of the same nonterminal in the abstract
                  syntax representation.

POTCommand := POTProgram | POTPrimitive | POTDeclaration || ;
  POTProgram := POTAndCommands |
                POTSelCommands |
                POTFirstCommands |
                POTWhileCommands |
                POTUntilCommands |
                POTSeqCommands || ;
    POTAndCommands := 'each POTCommand ↑ '; 'end |> POTCheckEllipsis ;
    POTSelCommands := 'choose { POTCriterion } 'from POTCommand ↑ '; 'end;
    POTFirstCommands := 'first 'of POTCommand ↑ '; 'end |> POTCheckEllipsis;
    POTWhileCommands := 'while POTCommand#
```

```
                    'do POTCommand ↑ '; 'end |> POTCheckEllipsis;
  POTUntilCommands := 'until POTCommand#
                    'do POTCommand ↑ '; 'end |> POTCheckEllipsis;
  POTSeqCommands := 'begin POTCommand ↑ '; 'end |> POTCheckEllipsis;
       POTInteger := LEXEME |> INTEGER? ;

POTPrimitive := POTPrimitiveCommand { 'by POTCommand } || ;
  POTPrimitiveCommand := POTUserCommand
                       | POTParsedCommand
                       | POTEllipsis
                       | POTLispCommand
                       | POTHistoryEvent || |> POTFillEvent ;
  POTUserCommand := LEXEME |> POTUserCommandFilter;
  POTLispCommand :=  POTCommandName { POTArgument + } ;
  POTParsedCommand := POTParsedName
                      { POTParsedArgument ↑ POTParsedSeparator } ;
  POTHistoryEvent := LEXEME |> HISTORYEVENT? ;
  POTEllipsis := ' ... ;
  NOTE the following production is actually a GLEXEME production which
 computes the grammar and delimiters for parsing.  See the "params" group
 in POPAR:-TRANSFORMATION-SYSTEM;
    POTParsedArgument := LEXEME ;
  NOTE separator below is obtained from the command formal parameter
      description;
      POTParsedSeparator := LEXEME |> POTSeparatorFilter;
    POTArgument := LEXEME |> LISPEXPRESSION? ;
  POTCriterion := POTInteger || ;

POTDeclaration := POTVariableDeclaration | POTCommandDeclaration || ;
  POTVariableDeclaration := 'local POTFormalParameter ↑ ', ;
    POTFormalParameter := POTVariableName { ': POTVariableName# } ;
  POTCommandDeclaration := POTCommandType POTCommandName
                           { '( { POTFormalParameter ↑ ', } ') }
                           '= POTCommand ;
    POTCommandType := 'command | 'metacommand ;
      POTVariableName := POTIdentifier || ;
      POTParsedName := LEXEME |> POTHasParsedArguments;
      POTCommandName := POTIdentifier || ;
        POTIdentifier := LEXEME |> IDENTIFIER? ;
```

BIBLIOGRAPHY

[Allen 75] Allen, F. E., *Bibliography on Program Optimization*, IBM Research, Yorktown Heights, New York, Technical Report RC 5767, 1975.

[Babich 78] Babich, W. A., and M. Jazayeri, "The method of attributes for data flow analysis: Parts I and II," *Acta Informatica* 10, (3), 1978, 245-264,265-272.

[Balzer 69] Balzer, R. M., "EXDAMS--extendable debugging and monitoring system," in *Spring Joint Computer Conference*, pp. 567-580, IFIP, 1969.

[Balzer 73] Balzer, R. M., *Language-independent Programmer's Interface*, USC/Information Sciences Institute, Technical Report RR-73-15, 1973.

[Balzer 76] Balzer, R., N. Goldman, and D. Wile, "On the transformational implementation approach to programming," in *Proceedings of the 2nd International Conference on Software Engineering*, pp. 337-334, IEEE, 1976.

[Barstow 79] Barstow, D. R., *Knowledge-based Program Construction*, Elsevier, North-Holland, 1979.

[Bauer 76] Bauer, F. L., "Programming as an evolutionary process," in *Proceedings of the 2nd International Conference on Software Engineering*, pp. 223-234, IEEE, 1976.

[Bauer 81] Bauer, F. L., M Broy, H. Partsch, P. Pepper, et al., *Report on a Wide Spectrum Language for Program Specification and Development*, Technische Universitaet Muenchen, Technical Report TUM-I8104, May 1981.

[Broy 81] Broy, M., and P. Pepper, "Program development as a formal activity," *IEEE Transactions on Software Engineering* 1, January 1981, 14-22.

[Burstall and Darlington 77] Burstall, R. M., and J. Darlington, "A transformation system for developing recursive programs," *Journal of the ACM* 24, (1), 1977, 44-67.

[Caine 75] Caine, S. H., and E. K. Gordon, "PDL--a tool for software development," in *National Computer Conference Proceedings, 1975*, AFIPS, 1975.

[Cameron 82] Cameron, R. D., and M. R. Ito, *Grammar-based Definition of Meta-programming Systems*, University of British Columbia, Vancouver, Technical Report, January 1982.

[Cheatham 79] Cheatham, T. E., G. H. Holloway, and J. A. Townley, "Symbolic evaluation and the analysis of programs," *IEEE Transactions on Software Engineering* 5, (4), July 1979, 402-417.

[Cheatham 81] Cheatham, T. E., G. H. Holloway, and J. A. Townley, "Program refinement by transformation," in *Proceedings of the 5th International Conference on Software Engineering*, pp. 430-437, IEEE, March 1981.

[Chiu 81] Chiu, W., *Structure Comparison and Semantic Interpretation of Differences*, Ph.D. thesis, University of Southern California, 1981.

[Darlington 79] Darlington, J., and M. Feather, *A Transformational Approach to Modification*, Imperial College, London, Technical Report 80/3, 1979.

[Darlington 82] Darlington, J., "The structured description of algorithm derivations," in *Algorithmic Languages: Proceedings of the IFIP TC-2 International Symposium*, North-Holland, 1982.

[Deutsch 73] Deutsch, L. P., *An Interactive Program Verifier*, Ph.D. thesis, University of California, Berkeley, June 1973.

[Dijkstra 76] Dijkstra, E. W., *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[Feather 79] Feather, M. S., *A System for Developing Programs by Transformation*, Ph.D. thesis, University of Edinburgh, Department of Artificial Intelligence, 1979.

[Feiler 80] Feiler, P. H., and R. Medina-More, *An Incremental Programming Environment*, Carnegie-Mellon University, Technical Report, April 1980.

[Fickas 80] Fickas, S., "Automatic goal-directed program transformation," in *Proceedings of the First Annual National Conference on Artificial Intelligence*, pp. 68-70, The American Association for Artificial Intelligence, 1980.

[Gerhart 75] Gerhart, S. L., "Knowledge about programs: A model and a case study," in *Proceedings of an International Conference on Reliable Software*, pp. 88-95, IEEE, 1975.

[Gerhart 80] Gerhart, S. L., et al., "An overview of ***Affirm***: A specification and verification system," in *Proceedings IFIP 80*, pp. 343-348, Australia, October 1980.

[Geschke 72] Geschke, C. M., *Global Program Optimizations*, Ph.D. thesis, Carnegie-Mellon University, 1972.

[Gordon 78] Gordon, M., R. Milner et al., "A metalanguage for interactive proof in LCF," in *Proceedings of a Conference Symposium on the Principles of Programming Languages, 1978*, pp. 119-130, 1978.

[Griss 76] Griss, C., M. Griss, and J. Marti, *META/LISP*, University of Utah, Utah Computational Physics Group, Technical Report Operating Note No. 24, 1976.

[Habermann 80] Habermann, A. N., "An overview of the Gandalf project," in *CMU Computer Science Research Review 1978-79*, Carnegie-Mellon University, 1980.

[Kahn 75] Donzeau-Gouge, V., G. Huet, G. Kahn, B. Lang, and J. J. Levy, "A structure-oriented program editor: A first step towards computer assisted programming," in *International Computing Symposium, 1975*, pp. 113-120, North-Holland, 1975.

[Kibler 78] Kibler, D. F., *Power, Efficiency, and Correctness of Transformation Systems*, Ph.D. thesis, University of California, Irvine, 1978.

[Knuth 70] Knuth, D. E., and P. B. Bendix, *Simple Word Problems in Universal Algebras*, Pergamon Press, New York, pp. 263-297, 1970.

[Loveman 77] Loveman, D. B., "Program improvement by source to source transformation," *Journal of the ACM* 24, (1), January 1977, 121-145.

[Low 74] Low, J. R., *Automatic Coding: Choice of Data Structures*, University of Rochester, Computer Science Department, Technical Report 1, 1974.

[Neighbors 80] Neighbors, J. M., *Software Construction Using Components*, Ph.D. thesis, University of California at Irvine, 1980.

[Partsch 81] Partsch, H., and R. Steinbrueggen, *A comprehensive survey on program transformation systems*, Technische Universitaet Muenchen, Technical Report TUM I8108, July 1981.

[Rich 81] Rich, C., "A formal representation for plans in the Programmer's Apprentice," in *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, University of British Columbia, August 1981.

[Schwartz 75] Schwartz, J. T., *On Programming, An Interim Report on the SETL Project*, New York University, Courant Institute of Mathematical Sciences, Technical Report, June 1975.

[Sintzoff 80] Sintzoff, M., "Suggestions for composing and specifying program design decisions," in *4th International Symposium on Programming*, Paris, April 1980.

[Standish 76] Standish, T. A., D. C. Harriman, D. F. Kibler, and J. M. Neighbors, "Improving and refining programs by program manipulation," in *ACM National Conference Proceedings*, pp. 509-516, ACM, 1976.

[Swartout 81] Swartout, W.R., "Explaining and justifying expert consulting programs," in *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, University of British Columbia, August 1981.

[Teitelbaum 81] Teitelbaum, T., and R. Reps, "The Cornell Program Synthesizer: A syntax-directed programming environment," *Communications of the ACM* 24, (9), September 1981, 563-573.

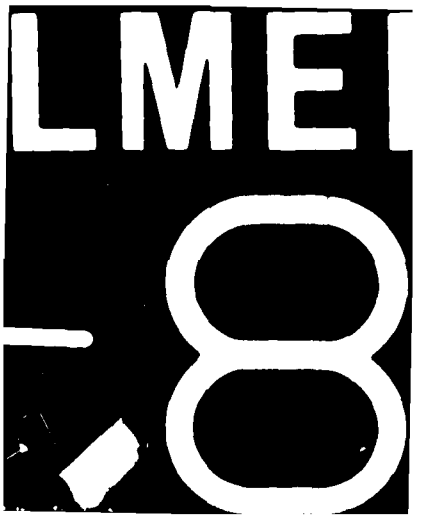
[Teitelman 78] Teitelman, W., *Interlisp Reference Manual*, Xerox Palo Alto Research Center, 1978.

[Waters 82] Waters, R. C., "The Programmer's Apprentice: Knowledge based program editing," *IEEE Transactions on Software Engineering* 8, (1), January 1982, 1-12.

[Wile 82] Wile, D. S., *POPART: Producer of Parsers and Related Tools, System Builder's Manual*, USC/Information Sciences Institute, TM-82-21, 1982.

[Wulf 75] Wulf, W. A., R. K. Johnsson, C. B. Weinstock, S. O. Hobbs, and C. M. Geschke, *The Design of an Optimizing Compiler*, American Elsevier, New York, 1975.

[Yonke 75] Yonke, M. D., *A Knowledgeable, Language-independent System for Program Construction and Modification*, USC/Information Sciences Institute, Technical Report RR-75-42, October 1975.


LMEI
.8